**Challenges and Advantages of Distributed Systems in Mainstream Computing:**

**Exploring The History and Future of Process Migration.**

by

Mark Zulli

Submitted to the Department of Mathematics and Computer Science

School of Liberal Arts and Sciences

in partial fulfillment of the requirements

for the degree of Bachelor of Arts

Purchase College

State University of New York

May 2020

Sponsor: Athar Abdul-Quader

Second Reader: Irina Shablinsky

Contents

**No table of contents entries found.**

**Abstract:**

Process Migration is a fundamental component of modern loosely-coupled distributed systems. Heterogeneous distributed systems in the High Performance Computing (HPC) services industry provide the foundation of the modern internet. Despite its success in large scale applications, modern distributed solutions for process sharing have not been brought to mainstream personal computers. As pervasive computing as a trend continues to grow, introducing process migration techniques in small scale environments promises several advantages. Local devices forming a processing grid would allow for greater utilization and optimization of finite resources, like mobile device battery life, CPU cycles, and file storage and enables process persistence. This work discusses the founding philosophies of modern distributed computing techniques as well as the history of distributed systems, their implementations and ways that pervasive and distributed computing will shape the digital future.

**Introduction:**

Distributed Computing is a broad field of study regarding systems whose components are located over a network. Typically this field is broken down into two distinctions, tightly coupled and loosely coupled systems. This distinction denotes whether memory is shared or not, respectively. This work focuses on the historical advancements that shaped loosely coupled distributed systems, where these systems are implemented today, and how these systems can be implemented in the future.

There are *seven* main areas of complication that must be considered when implementing distributed systems, both historically and currently:

*Non Determinism* - Essentially the detection and management of failure caused by either a node or the network “Without the ability to precisely reproduce a buggy execution, It is challenging for developers of distributed applications to track down the root cause of a bug.”(Hunt et al. 1)

*Heterogeneity* - systems must be able to adapt to different computing environments, including differing hardware devices, operating systems, networks, programming languages, and object structures.

*Transparency* - “the concealment from the user and the application programmer of the separation of components in a distributed system, so that the system is perceived as a whole rather than as a collection of independent components.” ([Bouchrika](https://www.ejbtutorial.com/distributed-systems/challenges-for-a-distributed-system))

*Openness* - New resource sharing services can be added and made available for use by a variety of client programs.

*Concurrency* - Applications and services shared in a distributed system must keep independent data consistent. Especially in the event of multiple nodes competing for finite resources.

*Security*- Distributed systems must employ strategies to protect against unauthorized access to resources, data corruption and attacks that prevent authorized users from accessing resources.

*Scalability* - The system should remain reliable and efficient independent of the number of users connected to the system.

**History:**

While the history of computing techniques provided is more detailed, it is important in understanding the development of the computer structures that enable distributed computing. Often innovation in the computing industry is driven by visionaries who dared think outside the confines of conventional technology, and their philosophies echo through generations of technology. (Xerox Parc) In this section, our aim is to gain knowledge to draw parallels between the state of computing at its conception and the state of distributed computing as a whole. Specifically with a focus on addressing what innovations and ideologies contributed to the development of distributed computing computing.

**Operating Systems:**

To understand the complexity of modern distributed computing systems, it is important to understand the basic design principles of operating systems in general.

An “Operating System” is a program that provides an interface between the user program and computer hardware. The necessity of a program-based interface between computer hardware and user programs was realized by General Motors for their IBM 704 Mainframe in 1956. Prior, computing was both too expensive a commodity to waste processing cycles on program scheduling. IBM and General Motors, developed single stream batch processing in their GMOS allowing programs to be submitted in groups and to be executed sequentially. GMOS was extremely simple, it only allowed for one process to be run at a time, and was implemented based off of the 3 phase design of industrial engineer Robert L Patrick. The three phases being input translation, computation and output translation.

Ths System allowed for optimization of the downtime created by slow I/O devices between processes, and allowed computer programmers, who were in short supply, to continue coding while their program ran. However this system also decreased programming efficiency. It was the obligation of the programmer to maintain all debugging code on their own; while a run time monitor would try to dump memory of failed programs for the programmer to examine, It was the responsibility of the programmer to implement this correctly. Despite lacking modern features, like a graphical user interface or time-sharing technology, this system introduced concepts like process IDs, separation of program data and trapping, that are still conventional for modern operating systems.

**Interrupts:**

Another modern operating systems concept that emerged was the *interrupt*. Interrupts are structures that allow for error handling as well as Input and Output operations. The interrupt is commonly credited to Dijkstra, based on his research on interfacing with the Electrologica X-1 system for the University of Amsterdam. However, interrupts existed in rudimentary capacity in the architecture of machines before it, previously defined as *Trapping*.

The IBM-704 mainframe, released in 1954, and some systems prior, had a hardware based trapping system to switch execution addresses to a pre-specified place in memory. The first documented instance of a trapping mechanism appears as early as 1953 in systems like the univac 1103 however manuals for such systems are largely lost to time. In the IBM mainframe, this was bit 0000. Trapping was executed either by the trp command in machine code or through controls on the terminal. This allowed programmers to control program flow or handle errors. It was the responsibility of the computer programmer to implement the trapping mechanism.

While the architecture of trapping may seem comparable to the mechanisms introduced by Djikstra, the problem Dijkstra sought to solve was extremely different. Dijkstra sought to implement a method for managing input and output (I/O) devices to allow for both concurrency and visibility. Dijkstra recalls his inspiration, the electronic typewriter. The device could be passed instructions from the processors output register to select the correct letter. However, the physical printing of the typewriter took orders of magnitude longer than calculation. This was a problem; without knowing when the typewriter had finished printing the letter, the next instruction could change the output register and potentially tangle the arms of the typewriter, damaging the expensive equipment. Simple solutions, like waiting a specific amount of time, were dependent on the clock speed of the system; what if the processor was twice as fast? Dykstra recognised that cheap code should not be able to break expensive hardware, and his small research team of 6 considered the problem of coordinating I/O and processing such that they may happen independently and visibly.

While working as a consultant with Electrologica for their X-1’s architecture, he implemented a hardware based monitor would report to the cpu when action was needed and temporarily halt the processor temporarily to locate and handle the urgent action, and then resume the action, allowing for I/O devices and processors to work concurrently. While this was significant in optimizing hardware, the new challenge was now coordinating such that this could function reliably. At its conception, this concept was largely unaccepted by Dijkstra because it introduced *non-determinism* into the system. There was no way to determine where the processor would stop in execution, and therefore irreplicable behavior would arise. This was solved by modifying other aspects of the X-1 system allowing for the detection of states in which the process could be safely halted.

Dijkstra recalls 3 fundamental problems that implementing the Interrupt solved. Firstly it allowed for processes to be switched. This would develop into modern multiprocessing and time sharing. Secondly, this enabled the development of process scopes. And lastly, it solved the logical problems caused by multiple (often slower) resources sharing the processing unit (Dijkstra 2000) birthing the possibility of distributed systems. His work served as the foundation for modern *system calls.* **Virtualization and TimeSharing**

The development of *Timesharing* and *Virtualization* are largely interconnected because they are results of the similar abstractions. Time sharing allows for processing time to be distributed among users while Virtualization enables multiple operating systems to share the same processor. (Oracle 1.1.2)

Time Sharing was first theorized by John Backus, Professor at the Massachusetts Institute of Technology (MIT) during the 1954 summer session. The computing strategy didn’t reach mainstream attention until John McCarthy, another MIT professor, published a memo introducing his project operating system called compatible time sharing system (CTSS). His research involved implementing magnetic tape drums in such a way that efficiently enabled program swapping, this technique for time-sharing that sparked the entire Boston area’s interest. (Ryan) Currently, systems could only handle one user at a time, with the more laborious option of batch processing. (Conroy) MIT researchers demonstrated the viability of time sharing on their IBM-709 Mainframe and (Project Mac) This innovation did not go unnoticed by the US Air Force.(Conroy) In 1963, MIT received two million dollars in government funding through ARPA to replace their existing mainframe and provide research to the federal government about concurrent computing technology. Dubbed “Project MAC,” which stood for Multiple Access Computer, they required custom hardware that would be able to support the concept, and reached out to both IBM and GE for production. (Conroy)

It wasn't until 1964 that IBM really began research into the production of the virtual machine on their own. They had just released the new S-360 mainframe, a technological innovation in itself due to its backwards compatibility and several different bit modes. A combination of both project release timing and Bell Labs’ increased interest in time sharing sparked the CP-40.(Conroy)

IBM created the CP-40 as an OS based solution to running virtual machines on the S-360. This was only a research operating system though, and shortly after the CP-67 was released. (Project Mac) Essentially a commercial version which featured the first user interaction focused operating system. This was revolutionary, as before programs simply executed and returned result as either a physical printout or to a digital display, with no option for interaction or mediation by the user. In addition, the CP-67 offered significant computational advantages. (IBM) (Conroy)

The CP-67 System specifically provided a triple threat of innovation; More addresses, More address spaces, and obfuscation of OS memory. In detail, the 24-32 bit variable system allowed for more addresses to be referenced making the address space larger. (IBM) On top of this, the system allowed for multiple address spaces, meaning that the address space could be swapped out per user, allowing isolation between user memory. This was made possible due to a new technology developed by IBM called the DAT Box, which provided Dynamic Address Translation. In short, this was a game changer. Instead of the system being equally divided between two concurrent environments, the resources of the mainframe were shared among users, significantly increasing efficiency.

Unfortunately, despite the cult culture that was emerging rapidly at MIT and the surrounding boston area, the next iteration mainframe did not have the same DAT box that enabled virtualization by default. This was largely due to new antitrust legislature which demanded that IBM unbundle its hardware and software, causing all sorts of problems. The S-370 was released in 1970 and was revolutionary in its own way, featuring hardware segregation and monolithic memory for the first time and by 1972 virtualization was reenabled on the IBM VM Operating system. The IBM 370 coined the term Hypervisor in its manual in reference to the technology enabling virtualization and instruction management.

The S-360 essentially gave birth to our modern understanding of a Type 1 Hypervisor, an operating-system-employed “middleware” that allows for the translation of system-calls from the guest user to the hardware. Although the specific hardware enabled virtualization, the implementation for time sharing was at the operating system level. And development continued without major revolution in Type 1 until the Intel Core architecture brought virtualization to the mainstream in the 1990s. The core architecture relied on a newer feature of x86 called rings of privilege. Allowing a Virtual Machine Monitor to run at the lowest ring of privilege in the operating system was added to the instruction set. (G4G)

**Distributed Computing:**

**ALTO:**

The earliest implementation of distributed personal computing comes from Xerox in the 1970s in their Alto workstations. The Xerox team had set out to innovate for the layman. Their goal was to create a system that could easily be implemented for office automation, with each user having their own -- easy to use – desk-top computer.

This was a farfetched concept for its time, computers still cost hundreds of thousands, and Xerox was a copier company with no prospect of becoming a computer company. Naturally computer companies were not seeking to join with Xerox out of fear of competition from more established manufacturers like IBM. However, Xerox was able to acquire the company “Scientific Data Systems“ (SDS) for about nine times their annual revenue in Xerox stock, nine hundred million dollars worth. This was a well documented mistake, SDS didn’t share a vision of a time sharing interactivity based system and the project was passed to a new division headed by Bob Taylor, former Project Mac and ARPA researcher. Using many of the government funded technologies he and his team developed, the first Alto systems were completed in 1972.

The Alto Systems revolutionized computer hardware and software, featuring for the first time a workstation with a screen that allowed the user to interact with the machine via keyboard and mouse. This user experience was largely enhanced by support for Ethernet networking of other Alto machines and a server that provided a file system for storage and designed to handle specific tasks like high quality printing. The operating system, dubbed “AltoOS '' had implemented remote access of terminals and a “world-swap” function in its operating system such that programs could be interchanged on the fly. Process-swapping was achieved by saving the processes entire state to disk allowing for the machine to be utilized by other programs, and inter-process communication was achieved through the filesystem.

While Xerox introduced a torrent of innovation and philosophies that would directly influence the Apple Computer, The Alto System was a total commercial flop and would not see widespread production. Xerox had hedged that prices of computer hardware components would drop wildly in price with scaled production, however the alto system and its iterations during the Xerox PARC still cost over $10,000 per unit and never saw scaled production outside the offices of Xerox PARC Employees. While commercially unsuccessful, this marked the transition out of the era of “Big Iron’s” room sized computers and the beginning of personal computing as we know it.

{https://dl.acm.org/doi/10.1145/12178.12186 16 or 17 mintues in}

**Accent:**

More significant development in distributed personal computing was concurrently happening in Carnegie Mellon’s Spice Project. In 1981, George Robertson and Richard Rashid unveiled their new communication oriented kernel system at the Symposium on Operating Systems Principles (SOSP), called Accent [1]. Unique to Accent was its “abstraction of communication between processes” (65). Accent OS introduced a message based architecture that allowed for several innovative abstractions.

Interprocess communication was achieved through *Ports*, with ownership being confined to one machine at a time, to avoid concurrency issues. In addition, The Implementation of layers of virtual memory allowed systems to separate program facing addresses from the kernel facing addresses, this enabled greater transparency. However the most notable innovation was its use of copy-on-write virtual memory management allowing for much of the costly overhard of copying an entire process when forking. Using page tables, multiple programs could share the same virtual memory addresses until a process tries to modify the memory. The memory, being protected, creates a page fault which creates a separate instance of the process to maintain separation of process data. Combining these abstractions, process migration was later added to the Accent kernel. Accent was only a research operating system and was later followed by its successor project Mach, which remains relevant in modern computing and can be found in Mac OSX and IOS even today.

**DEMOS/MP:**

The operating system DEMOS was a unix-like operating system developed concurrently to research in accent. DEMOS was developed in the Soviet Union at the Kurchatov Institute of Atomic Energy in, a research facility with the aim of developing nuclear technology; there is not much in academia about the original development of the DEMOS systems. Despite this, modifications to the DEMOS operating system enabling reliable process transfer are well documented. While process migration was implemented in the DEMOS system at its conception, Micheal Powell, a researcher at Carnegie Mellon University largely improved the system’s reliability by implementing location transparency and a redundancy system. This was achieved by implementing residual page tables after the process had been migrated such that inter process communication between nodes could still be achieved without the location of the process.

The DEMOS system was of interest to Carnegie Mellon specifically because it employed complete encapsulation of process data into a single structure. This allowed for much easier abstraction of process data location by the kernel, as the program could be seen as a single object entity. In short, it was an organized and easily modifiable kernel. DEMOS/MP introduced an abstraction called “Links” which created dynamic addresses for the retrieval of files from other networked machines; Links were managed entirely by the operating systems kernel and changed with the changing location of the process. In addition, it implemented a message queue for incoming kernel calls from other machines, allowing for delayed communications while the process was in transit or unavailable. Links were inherently versatile, with some lasting the entirety of the computer's run-time, while other links, like reply requests, could be quickly created and destroyed.

The philosophy driving the research on DEMOS/MP was more as a foundation for distributed computing technology and to demonstrate the prospect of dynamic load balancing systems, rather than to sell a commercial product, and so the DEMOS/MP operating system sees little recognition despite the innovation that this development added.

**Andrew:**

The Andrew File System (AFS), was another distributed system implemented in 1986 by Carnegie Mellon’s Spice project, which focused on a shared file system and data migration and featured remote program execution. The Andrew File System was designed to be functional rather than a research operating system. It was developed off of the Berkeley Distribution of Unix, which gave it greater heterogeneity than other research oriented operating systems, which were typically designed for specific hardware. It implemented the already existing ARPA networking technologies, like TCP and IP address mapping and its goal was to be distributed throughout the CMU campus to hundreds of sun machines.

The Andrew file system worked, unlike other operating systems, as primarily a database of files located on a cluster of servers, called “VICE” which allowed workstations to query for files to be sent to the local disk. The architecture of the AFS relied on server units being responsible for specific subtrees of the filesystem and relied on systems local caching in order to maintain integrity between distributions of the same file and to significantly reduce calls to the network. When files were modified, messages were sent from the workstation to the Andrew File System to notify other workstations that their cache had been invalidated. Vice-II even employed a method of callbacks, allowing invalidated cached files to be revalidated by the system in the event of unwarranted change. By implementing server clusters, the system was inherently scalable, as servers could be added to the cluster to accomodate more network traffic. In addition, AFS employed encryption, requiring users to login and receive keys from the server, which greatly improved security of files and processes moving over the network.

**Sprite OS:**

Sprite OS was an operating system that merged many of the philosophies of primitive distributed systems into a more robust process migrating system. It was developed by Fred Douglass at UC Berkeley in 1991 with the aim of increasing efficiency by identifying idle machines on the network and exploiting their processing power. It was a BSD-Unix inspired system that prioritized both concurrency and network transparency, as well as migration of processes. It was built around the philosophy that in office and university settings where personal computers were accessible, idle processing power was plentiful, however if a user occupies the terminal, they now are entitled to its full processing power and that of the network. Following this idea, Sprite implemented a two way process transfer system that consisted of migration and eviction.

Similar to Andrew, Sprite OS featured a distributed server based file system, with name transparency, meaning that files appeared the same whether located locally or on the distributed file system. Many of the system calls, including critical such as exec, make and pmake were reimplemented with lock systems for concurrency protection. Its RPC systems were modified for significantly greater bandwidth by fragmenting larger files into packet clusters. This enabled the Sprite OS to transfer significantly more complicated process structures over the network without high cost.

**Modern Underutilization of Distributed Systems:**

Google and Amazon both have implemented extraordinary distributed computing networks for both file storage and distributed computing workloads. All modern high-performance-computing environments, like those at Amazon and Google, rely on distributed systems to manage the massive amounts of data. Increasingly, these systems use cheap, commodity hardware, with examples like Amazon even ditching the X86 architecture for lightweight and efficient ARM cores. Despite the abundance of processing power in cheap abundant devices, the personal computer paradigms have not shifted to reflect their ubiquity in our modern environments. These lightweight and cheap processors, often bundled with commodity hardware, like smart appliances, gaming consoles, and mobile devices, should be optimized to account for finite resources.

In the case of smartphones, which have a limited and variable battery life, a joint study was done by Duke UCLA and Microsoft's research team. The team implemented a distributed computing operating system called MAUI based on the android platform, and serve, where code could be offloaded and executed remotely. They measured battery life and performance over wireless networks with varying response times. The study showed that employing distributed computing over wifi in smartphones reduced power consumption by 10x in low communication technologies, like facial recognition, and by about 20% in communication intensive processes, like gaming. All tests over wifi showed significantly better execution time than the smartphone’s capabilities alone.

Not only do I believe that these systems should be commercialized, I believe the system of process migration could be exploited for convenience of the user in digital centric environments. Process transfer could be used to create persistent processes that can be transferred from one household device to another seamlessly, optimizing processing power from the network of connected heterogeneous devices.

**Reflection:**

In the conception of this paper, I had the idea of creating an infrastructure for processes to be mobilized across these devices, for both convenience and optimization. My original proposed implementation required capturing process states and transferring that over network, while employing some sort of virtualization layer in order to retain address spaces despite the processes location over network. I was certain that systems that I described were unique as I had never seen such ideas expressed in modern environments, and scholarly articles for process migration and remote application execution were non-existent, besides patents from various companies describing specific components of process saving and restoring.

It was late in the semester that I discovered that this research had been going on under the term “distributed systems” since the 1960’s. I soon realized that most of the valuable information could be found through conference archives rather than academic journals. By understanding the culture of operating systems development, I was able to locate a substantial amount of history and implementation, machine reference manuals and transcripts peppered with the names of famous computer scientists like J.R.Licklidder and Doug Englebart. It was easy to get lost in the history of distributed systems and